Overview

This document describes the TI Graph library (TIGL) implementation on TI Keystone-II. The library defines a set of standard API's for developing graph algorithms on graphs of massive size. It provides a seamless interface to TI keystone-II hardware for single-core/multicore, and single-device/multi-device configurations.

The library is built using generic programming concepts that is based on template C++ classes. The provides maximum flexibility for the algorithm developer to define his customized data structure for his/her algorithms without impacting the core library.

The library supports the following hardware configuration on TI keystone-II:

1. Single Device/ Single ARM core.
2. Single Device/ Four ARM cores (using openMP).
3. Multiple Devices / Single ARM Core (using openMPI)
4. Multiple Devices / Four ARM cores (using openMPI + openMP)

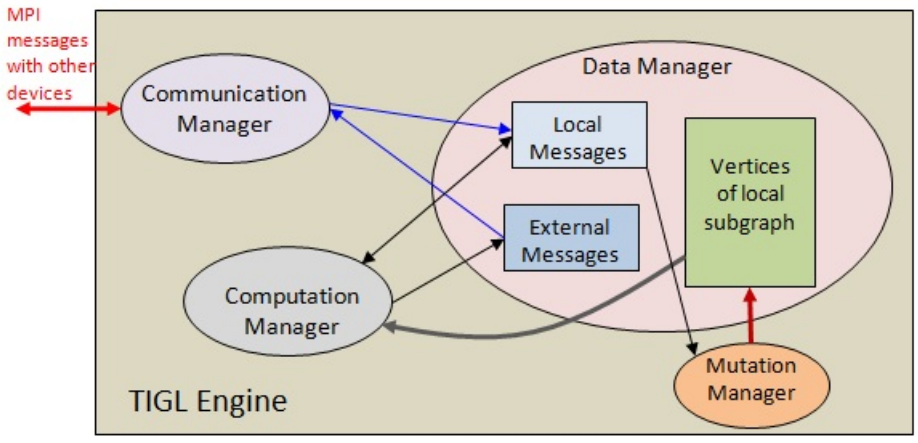
Background

This TI Graph Library (TIGL) is a software implemenation of the the Pregel model for graph processing that is developed by Google and is widely used as a scalable platform for graph processing. It is a message-based system that uses Bulk-Synchronization Parallel (BSP) model, where multiple devices perform local processing and exchange messages at a global timing barrier.
A graph algorithm is segmented into stages (called supersteps) that are executed sequentially and a global time barrier is deployed between successive stages to synchronize all workers (i.e., devices). Each worker (i.e., device) performs the computation for a subset of vertices in the graph, and workers communicate by messages between the vertices of each worker. At each superstep, each active vertex in each worker has a group of messages that are processed in the core compute function. The compute function produces a set of messages (usually to the children nodes) that are to be processed in the following superstep. The contents of the messages and the procedure for the compute function are dependent on the graph algorithm. In addition to the core compute function, the interface defines one other algorithm-dependent API’s: combine. The combine procedure aims at reducing the overall number of messages by combining messages going to a given destination vertex (usually using the sum or the max/min operators). Further, a general aggregate procedure collects global statistics of the algorithm and the graph that are shared by all vertices and could be used in the compute function. A graph algorithm terminates if the maximum number of supersteps is reached or if all vertices become inactive.

Software Modules

Overview

The main components of the TIGL library are summarized in the following figure.



The library was built using C++ object-oriented programming. We used generic programming concepts to enable the flexibility of the graph representation to handle any application without compromising the performance. Most classes are designed as template classes that can be customized to the application.

Therefore, most of the library is in the form of header files (.h and .hpp) that include both the definition and implementation of the template class.

On each device, the pseudo-code for executing the engine is as follows (from **tigl::run**):

```
//comment: initializing all engine components
dataManager->init();
communicationManager->init();

//comment: running all tasks
ComputeTask();
if(numDevices > 1)
{
    TxTask();
    RxTask();
}
```

The computeTask can be summarized the following pseudo code (from **tigl::computeTask**)

```
//comment: initializing all engine components
initSuperStep();                        // enable Tx/Rx
computationManager->init();
endSuperStep();                         // flush and disable Tx/Rx

while(ALGORITHM_NOT_COMPLETED)
{
    MPI_BARRIER();                              //comment: block synchronization point
    initSuperStep();                            // enable Tx/Rx
    mutationManager->processMutations();
    computationManager->superstep();            //comment: executing a single superstep
    endSuperStep();                             // flush and disable Tx/Rx
}
```

and the pseudo-code for the superstep execution (**tiglComputationManager::superstep**) is as follows

```
combineMessages();
initAggregateInfo();
clearMessageBuffers()
loop on active vertices
    vertex = getVertex();
    outMessages = execute(vertex, Messages(vertex))
    updateAggregateInfo(vertex,outMessages);                //comment: updating aggregate information
    distributeOutMessages(currOutMessages,countSS);
endloop
```

and the pseudo-code for distributeOutMessages is as follows

```
loop on output messages
    rcvr = getRcvr();
    if rcvr is local
        add to the local message buffer of rcvr
    else
        add to the external message buffer
endloop
```

The Rx task proceeds as follows (from **tiglCommunicationManager::continuousRecvMessages**)

```
while(Rx is Disabled)
    wait
end

while there exist messages to receive

    while existMessage2Read
        Start a Reading Thread
    end

    while existMessage2Process
        processSingleMessage(message)
    end
end // of the target messags
RxComplete = true;
```

Several non-blocking MPI read thread are activated in parallel to read the data/system/mutation messages from other devices.

Similarly, the Tx task proceeds as follows (from **tiglCommunicationManager::continuousSendMessages**)

```
while(Tx is Disabled)
    wait
end

while superstep is active
    while existMessage2Tx
        Start a Tx Thread
    end
end // of the target messags
send to each device the total number of sent messages to that device
```

```
TxComplete = true;
```

Data Structures

Base Classes

The two base classes for graph representation are the edge class and the vertex class. The edge class can have different interfaces depending on the graph topology. The basic edge class which is used with unweighted graphs is

```
template <typename idType>
class EdgeBase {
    idType node;  // the id of the target/src vertex
}
```

For weighted graphs, another field is added to define the weight

```
template<typename idType, typename dataType >
class EdgeW: public EdgeBase<idType> {
    dataType    edgeData; // edge weight information
}
```

The base vertex class that contains minimal vertex information is **VertexBase** which is defined as

```
template<typename DataType, typename AlgDataType, typename EdgeType, typename IdType >
class VertexBase{
    IdType          id;             // unique vertex ID
    DataType        vertexData;     // vertex data field
    AlgDataType     algData;        // Algorithm-specific data
    OUT_EDGE_BUF    outEdges;       // output edge buffer
}
```

To enable messaging between vertices during processing, few extra fields are added to define the core vertex class **Vertex** that inherits the **VertexBase** class

```
template<typename DataType, typename AlgDataType,typename EdgeType,typename InitType, typename IdType,
      typename MessageType>
class Vertex:
public VertexBase<DataType, AlgDataType, EdgeType, IdType>
{
    bool                    flagActive;         // active flag

public:
    virtual uint32 compute(MESSAGE_BUF_TYPE * pIn, LOCAL_MESSAGE_BUF * pOut, uint32 numMessages);
    virtual uint32 init(INIT_TYPE * initParams, LOCAL_MESSAGE_BUF * pOut);
}
```

The class has some more detalied attributes are described in the class documentation. Some of the class members are defined as as static because only a single copy of these attributes is needed for all objects. The class defines a virtual version of the two main API interface functions compute and init that need to be redefined by each graph algorithm.

Container Classes

In our implementation the graph is stored as an adjacency list, which is the list of vertices that make the graph along with their edges (the edges of each vertex are defined within the vertex object). The core container class is **SubgraphBase** whose data types are determined by the vertex type.

```
template <typename CVertex>
class SubgraphBase
{
    vector<CVertex> vertexList;     // physical storage of the subgraph
    POS_LIST_TYPE   vertexPos;      // the position of each vertex
    bool            flagDirected;   // directed/undirected graph
    uint32          posHashSize;    // the size of the position hash table
}
```

The vertices are stored in vertexList buffer. To accelerate random-access of vertices, we use another buffer vertexPos that contains a hash table for vertex positions. The vertices are hashed by their id, and an ordered collision table is maintained to hold vertices with the same hash value.

The core container **Subgraph** inherits the **SubgraphBase** class and adds the necessary message buffers for messages-based processing. It is defined as follows

```
template <typename CVertex, typename MessageType>
class Subgraph:
        public SubgraphBase<CVertex>
{
    //  ping-pong  data message buffers
    vector<MESSAGE_BUF_TYPE> * evenTimeMessages;
    vector<MESSAGE_BUF_TYPE> * oddTimeMessages;
```

```
    EXTERNAL_MESSAGE_BUF * externalOutMessages;//External Data messages buffer.

    vector<SysMessage>  outSysMessages; // output system message buffer

    // ping-pong Local system message buffer
    vector<SysMessage>  evenInSysMessages;
    vector<SysMessage>  oddInSysMessages;


    MUTATION_MBUF * outMutationMessages; //  buffer of external mutation messages
    MUTATION_MBUF * inMutationMessages;  //  Buffer for local mutation messages

}
```

Aggregate Information Container

The aggregate information is an integral part of the Pregel engine, and it is used by graph algorithms for monitoring. The class container for aggregate information **AggInfo** is defined as

```
template <typename AlgData>
class AggInfo
{
    uint32 numVertices;                             //total number of vertices in the graph
    uint32 numLocalVertices;                        //number of local vertices for this device
    uint32 countSS;                                 //superstep counter
    uint32 numTotalMessages[AGGREGATE_HISTORY];     //total number of output messages over a superstep
       history
    uint32 numExternalMessages[AGGREGATE_HISTORY]; //total number of external output messages over a
       superstep history
    uint32 numLocalActiveNodes[AGGREGATE_HISTORY]; //number of local active nodes over a superstep history
    uint32 numTotalActiveNodes[AGGREGATE_HISTORY]; //total number of active nodes over a superstep history
    AlgData maxActiveAlgData[AGGREGATE_HISTORY];    //maximum value of algorithm data of all active
       vertices over a superstep history
    AlgData minActiveAlgData[AGGREGATE_HISTORY];    //minimum value of algorithm data of all active
       vertices over a superstep history
    AlgData sumActiveAlgData[AGGREGATE_HISTORY];    //the sum of the values of algorithm data of all
       active vertices over a superstep history
}
```

The aggregate information includes basic statistics of the engine, including for example the number of active nodes and the number of output messages. It also contains some algorithm-specific information, e.g., the maximum and minimum value of the AlgData field of active vertices. The statistics are stored for a window of supersteps whose width is defined by AGGREGATE_HISTORY (which is set to 2 in the current release).

Computation Manager

The computation manager handles the vertex processing at each superstep. It executes the compute procedure for each active vertex, routes the output messages to the message buffers, combines local and external messages, and updates the aggregate information. It also has the initialization procedure that calls the init procedure of all local vertices. The main attributes of the compute manager class **tiglComputationManager** are :

```
template <typename SUBGRAPH_CLASS,typename COMBINER_TYPE, typename INIT_TYPE>
class tiglComputationManager
{
    vector<uint32> activeList;              // list of active vertices
    uint32 numLocalActive;                  // number of local active vertices
    AggInfo<ALG_DATA_TYPE> aggregateInfo;   // Aggregate Information object
    DATA_CONTAINER_CLASS * dataManager;     // pointer to the data manager object
    COMBINER_TYPE * combiner;               // pointer to combiner object
}
```

The activeList buffer is filled during the combine procedure where all nodes are checked for new messages. The core public functions of the **tiglComputationManager** are: **init** and **superstep** . The **init** procedure is called at the first superstep and has the form

```
init
for index = 1: numLocalVertices
    outMessages = verticesList(index).init()
    distributeOutMessages(outMessages)
    updateAggregateInformation(outMessages)
end local vertices loop
```

The **superstep** procedure is similar but it processes only the active nodes, and runs the execute procedure of each active vertex rather than the init procedure. Further, at the beginning a combine function is called to combine all local and external messages, that were not available at the previous superstep, are combined. The **superstep** core function proceeds as follows

superstep

```
CombineMessages()
processSystemMessages()
```

```
while activeList is not empty
    index = pop(activeList)
    outMessages = verticesList(index).execute       // comment: each vertex contains a pointer to its
       input message buffer
    distributeOutMessages(outMessages)
    updateAggregateInformation(outMessages)
end local vertices loop

if the activeList is empty
    return ALGORITHM_COMPLETED
endif
```

The function **distributeOutMessages** distributes external/local messages after vertex execution. In the current implementaiton, this function alone consumes approximately 50% of the overall execution time. A high level description is as follows

distributeOutMessages(messagesBuffer)

```
while messagesBuffer is not empty
    message = pop(messagesBuffer)
    rcvrVertex = message.getRecvr()
    if rcvrVertex is a local vertex
        push message to rcvrVertex messageBuffer of the next superstep
    else                    // rcvrVertex is in another device
        push to externalMessagesBuffer
    endif
end while
```

The last core function is **combineMessages** which combines input messages (both local and external) to each vertex prior to vertex execution.

combineMessages(messagesBuffer)

```
for index = 1: numLocalVertices
    if verticesList(index).hasMessages
        verticesList(index).combineMessages  // including combining from buffers of all threads
        push index to activeList
    else
        if verticesList(index).isActive
            push index to activeList
        endif
    endif
endfor
```

Communication Manager

The communication manager handles communication between devices. It is used only if more than one device is used. The core communication class is

```
template <typename DATA_CONTAINER_TYPE>
class tiglCommunicationManager
{
    DATA_CONTAINER_TYPE * dataPtr;  // pointer to the graph container that has all the message buffers
    uint32 numDevices;              // total number of devices
    uint64 totalNumVertices;        // total number of vertices in the whole graph
    uint32 rank;                    // rank of current device
}
```

The vertex processing is fully distributed and we do not have a controller device that oversees all the messaging. Therefore, some system messages are broadcasted to all devices to share the algorithm status. There are two buffers in the data manager for the input and output system messages, and we have a separate buffer for the output data messages. We use the non-blocking MPI calls MPI_Isend() and MPI_Irecv() to send/receive messages in TX and RX tasks respectively

The communication manager is also used to help the **Mutation Manager** in distributed construction of the graph (see **tigl::networkReadGraph**), where a single device reads the whole graph from a file and sends mutation messages to all other devices.

Mutation Manager

The mutation manager process all mutation messages during graph construction or during algorithm execution. The class definition is **tiglMutationManager**, which contains methods for all graph mutations.

Library Interface

The high-level interface class to the TIGL is the **tigl** class which performs all necessary allocations. The main attributes of the class are

```
template <typename ALG_VERTEX_TYPE, typename ALG_COMBINER>
class tigl
```

```
{
    uint32 rank;                            // rank of the device
    DATA_CONTAINER_TYPE * dataManager;      // The data manager object
    COMMUNICATOR_TYPE* communicationManager;// The communication manager
    COMPUTATION_TYPE * computationManager;  // the computation manager object
    ALG_COMBINER * combiner;                // the user-defined combiner object
    MUTATION_TYPE * mutationManager;        // the mutation manager
    uint32 numDevices;                      //total number of devices (1 if MPI is not used)
    uint32 numLocalVertices;                // number of local vertices
    bool flagDirected;                      // directed/undirected graph
    uint32 countSS;                         // superstep counter
}
```

A graph algorithm in the framework is fully parameterized by the definition of the vertex class and the combiner class. These are the two template parameters of the tigl class . The core procedure in this class is **run** which, as the name implies, runs the engine to execute the graph algorithm as defined by the vertex class. The interface class has also few simple I/O and profiling functions to load a graph from a file and evaluate the algorithm output (as well as the speed) as documented in the detailed library documentation

Example Algorithms

The Page Rank Algorithm

The page rank algorithm is a standard graph algorithm that assigns ranks to vertices in the graph. The formula for the page rank (PR) of a vertex u is expressed as

PR(u) = sum{v in N(u)} PR(v)/L(v)

where L(v) is the number of edges connected to vertex v and N(u) is the set of neighbor vertices to vertex u. The algorithm is computed recursively with the core compute function acts as

PR(u,t+1) = 0.15/num_vertices + 0.85*sum{v in N(u)} PR(v,t)/L(v)

where PR(v,t) is the rank of vertex v at superstep t. The convergence of the algorithm has been proven in the literature,

For this algorithm, the combine function sums the data value of all the messages from neighboring vertices.
The compute function has the form PR(v,t+1) = 0.15/num_vertices + 0.85*sum_messages

and this new page rank is broadcasted to all other neighbors as a message with the value PR(v,t+1)/L(v). The customized vertex definition for the page rank algorithm is **VertexP_pageR**. The algorithm is terminated after a predetermined number of supersteps is reached (set to 30 in our implementation).

The Single-Source Shortest Path Algorithm

The single-source shortest path problem aims at finding the shortest path between each vertex in the graph and a source vertex. In our implementation we use a variation of the standard Dijkstra algorithm. The messages that are exchanged between vertices contains the current distance between each vertex and the source. The algorithm proceeds as follows

```
for all active vertices
    for all received messages
        newDistance = min(oldDistance, messeageValue);

    broadcast newDistance to all neighbors;
    if the newDistance is less than globalMinDistance
        deactivate the vertex;
```

The combine function in this case is the minimum operator. The customized vertex definition for the single-source shortest path algorithm is **VertexP_SSSP**. The algorithm is terminated if there is no active nodes or if there is no new messages.

Compiling

The library is cross-compiled and tested on EVMK2H. The makefile that is distributed with the library has few flags that control the library options. The relevant flags are:

- *BUILD_OPENMP* = Yes/No : determines whether four ARM cores (Yes) or single ARM core (No) are used for building. It enables or disables the openMP interface.
- *BUILD_MPI* = Yes/No : determines whether multiple devices (Yes) or single device (No) are used for building. It enables or disables the MPI interface.
- *DEBUG_ENABLE* = No/Yes : determines whether detailed debug information is generated while running.
- *DEBUG_MPI* = No/Yes : determines whether MPI messaging debug information is generated while running.
- *PROFILE_ENABLE* = No/Yes : determines whether detailed profiling information is generated while running.
- *CMEM_ALLOC_ENABLE* = No/Yes : determines whether or not the CMEM module is used for memory allocation from

- *CMEM_ALLOC_ENABLE* = No/Yes : determines whether or not the CMEM module is used for memory allocation from MSMC memory area (more details are provided in the CMEM section)

The makefile would generate the corresponding executables in the output EVM directory which is specified by the environment variable *ARM_ROOT_DIR*

In the release, we included a general makefile "./build/MakefileLib" that contains necessary definitions and paths for compiling. This general makefile should be included in any customized makefile as illustrated by different examples.

Two executables are integral parts of the library. The first executable generates a graph with arbitrary number of vertices and edge factor (using the standard Rmat algorithm). The corresponding makefile is "./build/MakefileGenGraph". The second executable runs verification tests for the mutation manager. The corresponding makefile is "./build/MakefileMutationTester".

In addition, there are two separate makefiles for the two implemented graph algorithms: "./build/MakefilePageR" and "./build/MakefileSSSP".

General Procedure for Writing New Graph Algorithms

In the following, we show a step by step example of developing a graph algorithm using TIGL. We describe in details the Page Rank algorithm as described in an earlier section. The algorithm computes the rank of each vertex in the graph which is defined by the earlier **Formula**.

The first step in developing a new graph algorithm is defining the customized vertex class that inherits the core **Vertex** class. For the page rank algorithm the **algData** attribute holds the rank of each vertex, which is defined as double-precision floating number. The ID field of each vertex could be defined as uint32 or uint64 depending on the graph size. Each vertex also needs to know the total number of vertices in the graph; which is needed for the Page Rank computation.

The messages that are exchanged between neighbring vertices contain the page rank estimate of each vertex at the current superstep. Therefore, the customized message class is defined as

```
typedef Message<double,uint_fast32_t> DblMessage;
```

For unweighted graphs, the edge object is a simple customization of the core **EdgeBase** class.

In the customized **Vertex** class, we need to redefine the **init** and the **compute** functions. For the page rank algorithm, the initial value of the page rank of each vertex is set as 0.15/numVertices. The compute function implements the **recursive** page rank formula. Therefore, the customized vertex class **VertexP_pageR** can be defined as

```
   class VertexP_pageR:
        public Vertex<DataType,double,EdgeBase<uint_fast32_t>,uint_fast32_t,uint_fast32_t,DblMessage>
{
protected:
    static uint_fast32_t numVertices;   // the total number of vertices in the graph
public:
    uint_fast32_t compute(inputMessageBuffer,outMessageBuffer)
    {
        Iterate over all input messages
        {
            newRank += messageContent();
        }

        algData = newRank*0.85 + 0.15/(double)numVertices;

        Write algData to the outputMessageBuffer that contains messages to all neighbors

        if maximum number of supersteps is reached
            vote_to_halt();

        return the number of output messages;
    }


    uint_fast32_t init(outMessageBuffer)
    {
        algData = 0.15/(double)numVertices;

        Write algData to the outputMessageBuffer that contains messages to all neighbors
        return the number of output messages;
    }

};
```

The following step is to define the combiner class. In our case, the combine function simply sums the estimated page rank of all neighbors. Hence the customized combiner **tiglCombinerPageR** that inherits the core **tiglCombiner** class has the form:

```
class tiglCombinerPageR:  public tiglCombiner<DblMessage>
```

```
{
    inline uint32_t combineMessages(inputMessagesBuffer, outputMessagesBuffer)
    {
        return this->sumCombineMessages(inputMessagesBuffer,outputMessagesBuffer);
    }
};
```

At this point, a new algorithm, that can use the TIGL engine, has already been defined. To run the algorithm from the user program, the user needs to instantiate a tigl object with the customized class definitions of this algorithm as its parameters, i.e., define

```
tigl <VertexP_pageR<char>,tiglCombinerPageR>
                tiglInterface(DIRECTED_FLAG,NumVertices,deviceRank,numDevices);
```

and to execute the page rank algorithm within the user program (after loading the corresponding graph), a single command is called

```
tiglInterface.run(MAX_SUPERSTEPS, &initParams);
```

where initParams are simply the total number of vertices in the graph.

The final step is to write the makefile that defines the hardware configuration through the compiling flags as defined in the earlier section. For example, a sample makefile for generating the MPI version of the TIGL that executes the page rank algorithm looks as

```
BUILD_OPENMP = Yes
BUILD_MPI = Yes
DEBUG_ENABLE = No
DEBUG_MPI = No
PROFILE_ENABLE = No
CMEM_ALLOC_ENABLE = No
TESTER_ENABLE = No

include build/MakefileLib

OUTPUTFILE  =  <name of output file>


APPSRCFILE = <name of source file(s)>

APPHEADERFILES  = <name of algorithm-specific header file(s)>
CPP += <adding any additional include flags to the compile command>

$(OUTPUTFILE): $(APPSRCFILE)
    @$(CPP) $(APPSRCFILE)  -o $(OUTPUTFILE)
```